Network Programming with Sockets

CS 475

Anatomy of an Internet Connection

Client socket address
128.2.194.242:51213

Server socket address
208.216.181.15:80

Client host address
128.2.194.242

Server host address
208.216.181.15
Sockets

What is a socket?
- To the kernel, a socket is an endpoint of communication.
- To an application, a socket is a file descriptor that lets the application read/write from/to the network.
  - Remember: All Unix I/O devices, including networks, are modeled as files.

Clients and servers communicate with each other by reading from and writing to socket descriptors.

The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors.

Overview of the Sockets Interface

Client
- socket
- open_clientfd
- connect
- rio_readlineb
- rio_writen
- close

Server
- socket
- open_listenfd
- bind
- listen
- accept
- rio_readlineb
- rio_writen
- close

Await connection request from next client

Connection request
Socket Address Structures

Generic socket address:

- For address arguments to `connect`, `bind`, and `accept`.
- Necessary only because C did not have generic (void *) pointers when the sockets interface was designed.

```c
struct sockaddr {
    unsigned short  sa_family;    /* protocol family */
    char            sa_data[14];  /* address data. */
};
```

- `sa_family`: Family Specific

---

Socket Address Structures

Internet-specific socket address:

- Must cast `(sockaddr_in *) to (sockaddr *)` for `connect`, `bind`, and `accept`.

```c
struct sockaddr_in  {
    unsigned short  sin_family;  /* address family (always AF_INET) */
    unsigned short  sin_port;    /* port num in network byte order */
    struct in_addr  sin_addr;    /* IP addr in network byte order */
    unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```

- `sin_port`, `sin_addr`: Family Specific
- `sin_family`: 0 0 0 0 0 0 0 0
Example: Echo Client and Server

On Server

```
bass> echoserver 5000
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 4 bytes: 123
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 7 bytes: 456789
...```

On Client

```
kittyhawk> echoclient bass 5000
Please enter msg: 123
Echo from server: 123

kittyhawk> echoclient bass 5000
Please enter msg: 456789
Echo from server: 456789
kittyhawk>
```

---

Echo Client Main Routine

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];
    rio_t rio;
    host = argv[1];  port = atoi(argv[2]);
    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);
    printf("type:"); fflush(stdout);
    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, clientfd);
        printf("echo: ");
        Fputs(buf, stdout);
        printf("type: "); fflush(stdout);
    }
    Close(clientfd);
    exit(0);
}
```
Overview of the Sockets Interface

Client
- socket
- connect

Server
- socket
- bind
- listen
- accept

open_clientfd
- Connection request

Echo Client: open_clientfd

```c
int open_clientfd(char *hostname, int port) {
    int clientfd;
    struct hostent *hp;
    struct sockaddr_in serveraddr;
    if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1; /* check errno for cause of error */
    /* Fill in the server's IP address and port */
    if ((hp = gethostbyname(hostname)) == NULL)
        return -2; /* check h_errno for cause of error */
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    bcopy((char *) hp->h_addr_list[0],
          (char *) &serveraddr.sin_addr.s_addr, hp->h_length);
    serveraddr.sin_port = htons(port);
    /* Establish a connection with the server */
    if (connect(clientfd, (SA *) &serveraddr,
                  sizeof(serveraddr)) < 0)
        return -1;
    return clientfd;
}
```

This function opens a connection from the client to the server at hostname:port

Create socket
Create address
Establish connection
**Echo Client: open_clientfd**

*socket creates a socket descriptor on the client*
- Just allocates & initializes some internal data structures
- **AF_INET**: indicates that the socket is associated with Internet protocols.
- **SOCK_STREAM**: selects a reliable byte stream connection
  - Provided by TCP

```c
int clientfd; /* socket descriptor */
if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1; /* check errno for cause of error */
... (more)
```

---

**Echo Client: open_clientfd**

*open_clientfd (gethostbyname)*

The client then builds the server’s Internet address.

```c
int clientfd; /* socket descriptor */
struct hostent *hp; /* DNS host entry */
struct sockaddr_in serveraddr; /* server’s IP address */
...
/* fill in the server’s IP address and port */
if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(port);
bcopy((char *)hp->h_addr_list[0],
     (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
```

---

Check this out!
A Careful Look at bcopy Arguments

/* DNS host entry structure */
struct hostent {
  ...
  int h_length; /* length of an address, in bytes */
  char **h_addr_list; /* null-terminated array of in_addr structs */
};

struct sockaddr_in {
  struct in_addr sin_addr; /* IP addr in network byte order */
  /* Internet address structure */
  struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
  };
};

struct hostent *hp; /* DNS host entry */
struct sockaddr_in serveraddr; /* server’s IP address */
...
bcopy((char *)hp->h_addr_list[0], /* src, dest */
      (char *)&serveraddr.sin_addr.s_addr, hp->h_length);

Echo Client: open_clientfd (connect)

Finally the client creates a connection with the server.

- Client process suspends (blocks) until the connection is created.
- After resuming, the client is ready to begin exchanging messages
  with the server via Unix I/O calls on descriptor clientfd.

int clientfd; /* socket descriptor */
struct sockaddr_in serveraddr; /* server address */
typedef struct sockaddr SA; /* generic sockaddr */
...
/* Establish a connection with the server */
if (connect(clientfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0) {
  return -1;
}
return clientfd;
**Echo Server: Main Routine**

```c
int main(int argc, char **argv) {
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    struct hostent *hp;
    char *haddrp;

    port = atoi(argv[1]); /* the server listens on a port passed
    on the command line */
    listenfd = open_listenfd(port);

    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                           sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        haddrp = inet_ntoa(clientaddr.sin_addr);
        printf("server connected to %s (%s)\n", hp->h_name, haddrp);
        echo(connfd);
        Close(connfd);
    }
}
```

---

**Overview of the Sockets Interface**

- **Client**
  - `socket`
  - `bind`
  - `listen`
  - `connect`

- **Server**
  - `socket`
  - `bind`
  - `listen`
  - `accept`

- `open_clientfd` → `open_listenfd`
Echo Server: open_listenfd

```c
int open_listenfd(int port)
{
    int listenfd, optval=1;
    struct sockaddr_in serveraddr;

    /* Create a socket descriptor */
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    /* Eliminates "Address already in use" error from bind. */
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int)) < 0)
        return -1;

    /* Listenfd will be an endpoint for all requests to port
     * on any IP address for this host */
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons((unsigned short)port);
    if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
        return -1;

    /* Make it a listening socket ready to accept
     * connection requests */
    if (listen(listenfd, LISTENQ) < 0)
        return -1;

    return listenfd;
}
```
Echo Server: open_listenfd (socket)

socket creates a socket descriptor on the server.
  - AF_INET: indicates that the socket is associated with Internet protocols.
  - SOCK_STREAM: selects a reliable byte stream connection (TCP)

```c
int listenfd; /* listening socket descriptor */

/* Create a socket descriptor */
if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1;
```

---

Echo Server: open_listenfd (setsockopt)

The socket can be given some attributes.

```c
... /* Eliminates "Address already in use" error from bind(). */
if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
              (const void *)&optval , sizeof(int)) < 0)
    return -1;
```

Handy trick that allows us to rerun the server immediately after we kill it.

- Otherwise we would have to wait about 15 secs.
- Eliminates “Address already in use” error from bind().

Strongly suggest you do this for all your servers to simplify debugging.
Echo Server: open_listenfd (initialize socket address)

Initialize socket with server port number
accept connection from any IP address

```c
struct sockaddr_in serveraddr; /* server's socket addr */
... /* listenfd will be an endpoint for all requests to port
on any IP address for this host */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons((unsigned short)port);
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

<table>
<thead>
<tr>
<th>sin_port</th>
<th>sin_addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF_INET</td>
<td>INADDR_ANY</td>
</tr>
</tbody>
</table>

sin_family

IP addr and port stored in network (big-endian) byte order

---

Echo Server: open_listenfd (bind)

bind associates the socket with the socket address we just created.

```c
int listenfd;                  /* listening socket */
struct sockaddr_in serveraddr; /* server's socket addr */
...
/* listenfd will be an endpoint for all requests to port
on any IP address for this host */
if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
    return -1;
```
Echo Server: \texttt{open\_listenfd} (\texttt{listen})

\texttt{listen} indicates that this socket will accept connection (\texttt{connect}) requests from clients

\texttt{LISTENQ} is constant indicating how many pending requests allowed

\begin{verbatim}
int listenfd; /* listening socket */

.../* Make it a listening socket ready to accept connection requests */
if (listen(listenfd, LISTENQ) < 0)
  return -1;
return listenfd;
\end{verbatim}

We’re finally ready to enter the main server loop that accepts and processes client connection requests.

---

Echo Server: Main Loop

The server loops endlessly, waiting for connection requests, then reading input from the client, and echoing the input back to the client.

\begin{verbatim}
main() {
  /* create and configure the listening socket */
  while(1) {
    /* Accept(): wait for a connection request */
    /* echo(): read and echo input lines from client til EOF */
    /* Close(): close the connection */
  }
}
\end{verbatim}
Overview of the Sockets Interface

Echo Server: accept

accept() blocks waiting for a connection request.

```c
int listenfd; /* listening descriptor */
int connfd; /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
```

accept returns a **connected descriptor** (**connfd**) with the same properties as the **listening descriptor** (**listenfd**):
- Returns when the connection between client and server is created and ready for I/O transfers.
- All I/O with the client will be done via the connected socket.

accept also fills in client's IP address.
### Echo Server: `accept` Illustrated

1. **Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`.**

2. **Client makes connection request by calling and blocking in `connect`.**

3. **Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`.**

---

### Connected vs. Listening Descriptors

**Listening descriptor**
- End point for client connection requests.
- Created once and exists for lifetime of the server.

**Connected descriptor**
- End point of the connection between client and server.
- A new descriptor is created each time the server accepts a connection request from a client.
- Exists only as long as it takes to service client.

**Why the distinction?**
- Allows for concurrent servers that can communicate over many client connections simultaneously.
  - E.g., Each time we receive a new request, we fork a child to handle the request.
Echo Server: Identifying the Client

The server can determine the domain name and IP address of the client.

```c
struct hostent *hp;  /* pointer to DNS host entry */
char *haddrp;        /* pointer to dotted decimal string */

hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
    sizeof(clientaddr.sin_addr.s_addr), AF_INET);
haddrp = inet_ntoa(clientaddr.sin_addr);
printf("server connected to %s (%s)\n", hp->h_name, haddrp);
```

Echo Server: `echo`

The server uses RIO to read and echo text lines until EOF (end-of-file) is encountered.

- **EOF notification** caused by client calling
  `close(clientfd)`.
- **IMPORTANT:** EOF is a condition, not a particular data byte.

```c
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;
    
    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        upper_case(buf);
        Rio_writen(connfd, buf, n);
    }
    printf("server received %d bytes\n", n);
}
```
The RIO Package

RIO is a set of wrappers that provide efficient and robust I/O in apps, such as network programs that are subject to short counts

RIO provides two different kinds of functions
- Unbuffered input and output of binary data
  - rio_readn and rio_writen
- Buffered input of binary data and text lines
  - rio_readlineb and rio_readmb
  - Buffered RIO routines are thread-safe and can be interleaved arbitrarily on the same descriptor

Included in file csapp.c, csapp.h provided with Assignment 3

Unbuffered RIO Input and Output

Same interface as Unix read and write
Especially useful for transferring data on network sockets

#include "csapp.h"

ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);

Return: num. bytes transferred if OK, 0 on EOF (rio_readn only), -1 on error

- rio_readn returns short count only it encounters EOF.
  - Only use it when you know how many bytes to read
- rio_writen never returns a short count.
- Calls to rio_readn and rio_writen can be interleaved arbitrarily on the same descriptor.
RIO Example

Copying the lines of a text file from standard input to standard output

```c
#include "csapp.h"

int main(int argc, char **argv)
{
    int n;
    rio_t rio;
    char buf[MAXLINE];

    Rio_readinitb(&rio, STDIN_FILENO);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
    {
        Rio_writen(STDOUT_FILENO, buf, n);
    }
    exit(0);
}
```

Standard I/O: Reading Files

Reading a file copies bytes from the current file position to memory, and then updates file position

```c
char buf[512];
int fd;    /* file descriptor */
int nbytes;  /* number of bytes read */

/* Open file fd ... */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}
```

**Returns number of bytes read from file fd into buf**

- Return type `ssize_t` is signed integer
- `nbytes < 0` indicates that an error occurred.
- `short counts` (nbytes < sizeof(buf)) are possible and are not errors!
Motivation for RIO: Dealing with Short Counts

Short counts can occur in these situations:
- Encountering (end-of-file) EOF on reads
- Reading text lines from a terminal
- Reading and writing network sockets or Unix pipes

Short counts never occur in these situations:
- Reading from disk files (except for EOF)
- Writing to disk files

One way to deal with short counts in your code:
- Use the RIO (Robust I/O) package

Testing Servers Using telnet

The telnet program is invaluable for testing servers that transmit ASCII strings over Internet connections
- Our simple echo server
- Web servers
- Mail servers

Usage:
- unix> telnet <host> <portnumber>
- Creates a connection with a server running on <host> and listening on port <portnumber>.
Testing the Echo Server With `telnet`

```
bass> echoserver 5000
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 5 bytes: 123
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 8 bytes: 456789

kittyhawk> telnet bass 5000
Trying 128.2.222.85...
Connected to BASS.CMCL.CS.CMU.EDU.
Escape character is '^]'.
123
Connection closed by foreign host.
kittyhawk> telnet bass 5000
Trying 128.2.222.85...
Connected to BASS.CMCL.CS.CMU.EDU.
Escape character is '^]'.
456789
Connection closed by foreign host.
kittyhawk>
```